Foundations of Computer Science

Supervision 2

4. More on lists

4.1. Conceptual questions

1. Suppose we use a variant of the list zipping function which takes the two list arguments as a pair and has the type `zip : 'a list * 'b list -> ('a * 'b) list`. This way, the functions `zip` and `unzip` seem like they are each other's opposites. Mathematically, two functions $f : A \rightarrow B$ and $g : B \rightarrow A$ are inverses of each other if for all arguments $a \in A$, $g(f(a)) = a$, and if for all arguments $b \in B$, $f(g(b)) = b$. If only the first condition holds, we call g the *left inverse* of f, and if only the second condition holds, we call g the *right inverse* of f. Is `unzip` the inverse, right inverse or left inverse of `zip`? Justify your answer.

2. We know nothing about the functions `f` and `g` other than their polymorphic types:

   ```
   > val f : 'a * 'b -> 'b * 'a = <fun>
   > val g : 'a -> 'a list = <fun>
   ```

   Suppose that `f (1, true)` and `g 0` are evaluated and return their results. State, with reasons, what you think the resulting *values* will be, and how the functions can be defined. Can any of the definitions be changed if termination wasn't a requirements?

4.2. Exercises

3. a) Use the `member` function on Page 31 to implement the function `inter` that calculates the *intersection* of two OCaml lists – that is, `inter xs ys` returns the list of elements that occur both in `xs` and `ys`. You may assume `xs` and `ys` have no repeated elements.

   b) Similarly, code a function to implement set union. It should avoid introducing repetitions, for example the union of the lists `[4;7;1]` and `[6;4;7]` should be `[1;6;4;7]` (though the order should not matter).

4. Code a function that takes a list of integers and returns two lists, the first consisting of all nonnegative numbers found in the input and the second consisting of all the negative numbers. How would you adapt this function so it can be used to implement a sorting algorithm?

4.3. Optional questions

5. How does this version of `zip` differ from the one in the course?

   ```
   let rec zip xs ys = match xs, ys with
     | (x::xs, y::ys) -> (x,y) :: zip (xs,ys)
     | ([], [])       -> []
   ```

6. What assumptions do the "making change" functions make about the variables `till` and `amt`? What could happen if these assumptions were violated?

5. Sorting

5.1. Conceptual questions

1. You are given a long list of integers. Would you rather:
   a) Find a single element with linear search or sort the list and use binary search?
   b) Find a lot of elements with linear search or sort the list and use binary search?
   c) Find all duplicates by pairwise comparison or sort the list and check for adjacent values?
2. Another sorting algorithm (*bubble sort*) consists of looking at adjacent pairs of elements, exchanging them if they are out of order and repeating this process until no more exchanges are possible. Analyse the time complexity of this approach.

5.2. Exercises

3. Implement bubble sort in OCaml.

5.3. Optional questions

4. Another sorting algorithm (selection sort) consists of looking at the elements to be sorted, identifying and removing a minimal element, which is placed at the head of the result. The tail is obtained by recursively sorting the remaining elements.
   a) State, with justification, the time complexity of this approach.
   b) Implement selection sort using OCaml.

6. Datatypes and trees

6.1. Conceptual questions

1. Examine the following function declaration. What does `ftree (1,n)` accomplish?

```
let rec ftree k = function
  | 0 -> Lf
  | n -> Br(k, ftree (2*k) (n-1), ftree (2*k+1) (n-1))
```

6.2. Exercises

2. Write an function taking a binary tree labelled with integers and returning their sum.
3. Give the declaration of an OCaml datatype for arithmetic expressions that have the following possible shapes: floats, variables (represented by strings), or expressions of the form

$$-E \quad \text{or} \quad E+E \quad \text{or} \quad E\times E$$

Hint: recall how expressions differ from statements, and how their characteristic structure could be captured as a data type. It's a lot simpler than it may seem!

4. Continuing the previous exercise, write a function `eval : expr -> float` that evaluates an expression. If the expression contains any variables, your function should raise an exception indicating the variable name.

6.3. Optional questions

5. Prove the inequality involving the depth and size of a binary tree `t` from Page 53.

$$\forall \text{ trees } \mathtt{t}.\ \mathrm{count}(\mathtt{t}) \leq 2^{\mathrm{depth}(\mathtt{t})} - 1$$

   Hint: One way to do this is with a generalisation of mathematical induction called *structural induction*, where you analyse the *shape* (top-level constructor) of the tree `t`, and prove the property for the base case (leaf) and recursive case (branch). You can also try standard mathematical induction on some numerical property of the tree, but be careful with that you are assuming and what you are proving!

6. Give a declaration of the data type `day` for the days of the week. Comment on the practicality of such a datatype in a calendar application.

7. Write a function `dayFromDate : int -> int -> int -> day` which calculates the day of the week of a date given as integers. For example, `dayFromDate 2020 10 13` would evaluate to `Tuesday` (or whatever encoding you used in your definition of `day` above). *Hint*: Using Zeller's Rule might be the easiest approach.